Processing Object-Orientation-based Direction Queries in Spatial Databases *

Xuan Liu, Shashi Shekhar, Sanjay Chawla

Computer Science Department, University of Minnesota

EE/CS 4-192, 200 Union St. SE., Minneapolis, MN 55455

telephone: (612)624-8307

[*xliu*|*shekhar*|*chawla*]@*cs.umn.edu*

http://www.cs.umn.edu/research/shashi-group

Abstract

Direction based spatial relationships are critical in many domains including geographic information systems(GIS) and image interpretation. They are also frequently used as selection conditions in spatial queries. In this paper, we explore processing of queries based on object-orientation-based directional relationships. A new Open Shape based strategy(OSS) is proposed. OSS converts the processing of the direction predicates to the processing of topological operations between open shapes and closed geometry objects. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, and also eliminating the computation related to the world boundary. We perform algebraic analysis as well as experimental evaluation for OSS. The experimental result demonstrates that the OSS consistently outperforms classical range query strategy both in I/O and CPU cost.

Keywords: Direction, Orientation, Open shape, Topological operations, Spatial query processing, Range query.

*This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

1 Introduction

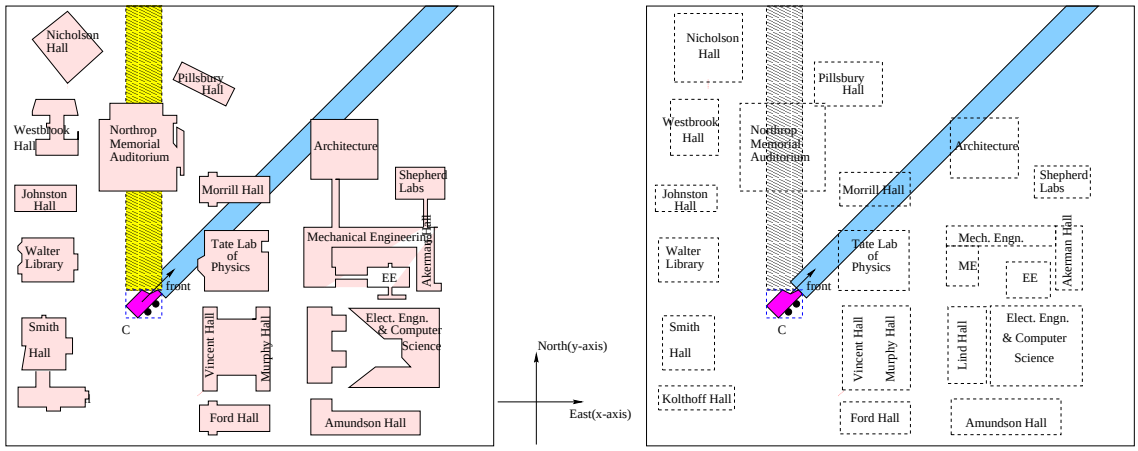
"Direction" is a common spatial concept frequently used as a selection condition in spatial databases[9, 1] or for similarity accessing in image databases[16]. Examples of direction queries used in army battlefield visualization[8] are "List the enemy targets to the *north* of Building B","Is there anything *over* the ridge?", and "Describe all tanks to the *left* of Landmark L". The first example refers to an absolute directional relationship(*north*) with respect to Building B, the second involves a viewer-orientation-based directional relationship, i.e. the direction is based on the orientation of the viewer, and the third example uses an object-orientation-based direction as its selection criteria.

It is important for a spatial database system to provide a mechanism for modeling and processing direction queries. Most of the previous work [2, 22, 14] on direction query processing has focused on processing predicates of absolute directions(e.g. North, East) using range query strategies. However, object-orientation-based direction queries and predicates(e.g. left) depend on the orientation of reference objects, which may be different from the global reference system. Because of this characteristic, the strategies that are efficient for absolute directions may be inefficient when used for object-orientation-based directions.

In this paper, we focus on the processing of spatial queries involving object-orientation-based direction predicates in selection conditions. Based on direction object model proposed in our recent work [20], we define a new abstract data type(ADT) for open shapes and propose a new Open Shape based strategy(**OSS**). OSS transforms the computation of the direction predicates into the computation of topological operations between open shapes and objects. Algebraic analysis and experimental evaluation demonstrate that OSS consistently outperforms classical strategies both in I/O and CPU cost.

1.1 Problem Definition

Absolute direction of a target object relative to a reference object is defined by their locations in the embedding space. Usually, the absolute direction is described in context of a global 2-D coordinate system e.g. (north up, east right) as shown in figure 1. Some examples of absolute directional relationships are $North$, $East$, and $NorthEast$. Figure 1(a) shows a small part of the campus map of University of Minnesota with a car C. A query involving an absolute direction predicate is "List all buildings north of C". The dashed shaded rectangle in the figure shows the region satisfying the query selection condition $North$. This shaded rectangle is called *direction region*, which consists exactly of all the points



(a) Campus map of University of Minnesota (b) Approximate buildings by MBRs

Figure 1: Absolute direction vs. Object-orientation-based direction

satisfying the direction predicate. The query identifies "Northrop Memorial Auditorium" as the only building satisfying the query.

Object-orientation-based direction of a target object is described with respect to the orientation of a reference object. The orientation of the target object is not relevant. In figure 2, the desk is an oriented object, but the flag is not. The flag is to the right of the desk based on the orientation of the desk. In other words, imagine the reference object *desk* were a person, then object-orientation-based direction is described with respect to the person's orientation and location.

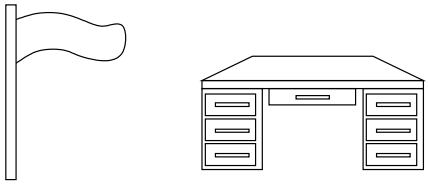


Figure 2: Object-orientation-based direction: The flag is to the right of the desk, assuming front of desk is facing out of the paper

In Figure 1(a), the car C has an intrinsic orientation described by the direction $front$. Consider a query based on the orientation of C, i.e., object-orientation-based direction query, "List all the buildings in front of the car C". The solid shaded region shows the direction region for "in front of the car C", and hence, the buildings in front of the car include "Tate lab", "Morrill Hall", and "Architecture".

It is common to approximate 2-D regions by minimum bounding rectangles(MBRs) which are or-

thogonal with respect to the global coordinate system. We refer to this type of MBR as MBR_g. Figure 1(b) replaces all the buildings in Figure 1(a) by their corresponding $MBR_g s$. However, for a reference object, the MBR may need to be taken based on the orientation of the object. We represent this type of MBR as MBR_o. Figure 3 shows examples of these two MBR types for a given object. The dashed orthogonal rectangle is the MBR according to the global coordinate system, and the solid rectangle is the MBR according to the orientation of the object.

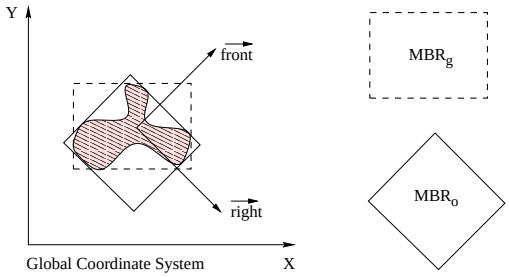


Figure 3: Different MBRs according to different coordinate systems

Collections of $MBR_g s$ can be organized by a R-tree for efficient search. Readers not familiar with R-trees may refer to related literature [23, 10, 5]. In the rest of the paper, we assume data objects are approximated by MBR_g and reference objects are approximated by MBR_o. For simplicity, if not explicitly clarified, MBR represents MBR_g.

In this paper, we intend to explore processing strategies for spatial queries which involve object-orientation-based direction predicates as selection criteria. The problem is denoted as Object-Orientation-based Direction Queries(OODQ) and can be formally defined as follows:

Object-Orientation-based Direction Queries(OODQ) problem:

Given: (a) A query involving an object-orientation-based direction predicate in selection conditions

(b) a R-tree index of the dataset

Find: All objects satisfying the selection condition

Objective: Minimize the number of page access.

Constraint: (a) Objects are approximated as MBR_g.

(b) There is only one R-tree index based on global coordinate system

1.2 Related work and our contributions

Direction is often modeled as a binary relationship between objects [3, 18, 6, 24, 1, 15, 7, 4, 19, 13]. Different direction predicate sets [14, 6, 11, 3] have been proposed using binary relationship view of directions. Research on processing direction-based queries has explored indexing methods and processing strategies. A previous study[22] evaluated different indices for processing absolute direction queries. Its results showed that spatial index (e.g. R-tree) is efficient for retrieving queries with constraints on both dimensions. Range query strategy(RQS) was used to implement queries involving absolute directions[2, 22, 14]. For example, consider the query based on the objects in Figure 4: *Find all objects that are exactly east of object B*. In the context of absolute directions, the x-axis and y-axis of the coordinate system may be aligned with the *East* and *North* directions. The direction queries based on absolute directions(B's East) can be viewed as a special case of range query as shown in figure 4. Here,

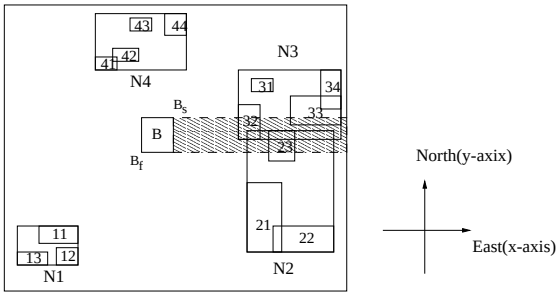


Figure 4: Processing absolute direction query using range query strategy

all the objects whose interiors intersect with the interior of the shaded *direction region* belong to the query result. Since an absolute direction is based on the global reference system, its direction region is a rectangle with sides parallel to the axes of the global coordinate system. It is natural and efficient to use range query strategy[22], which is specified by Algorithm 1 named RQS_AD. RQS_AD finds all object MBRs satisfying a given direction predicate using a R-tree. The input of RQS_AD is a set of data objects indexed by a R-tree and a direction region(*dirRectangle*). This algorithm starts from the root node of R-tree, excludes the intermediate nodes whose MBRs do not intersect with *dirRectangle*, and recursively searches the children of the remaining intermediate nodes. A refinement using the exact geometry of these objects may occur. But for simplicity, we will omit that in the rest of the paper.

Let us now consider extension of RQS_AD for object-orientation-based direction queries. If the orientation of the reference object is not aligned with the global coordinate system, then the corresponding direction region will not have sides parallel to the axes of the global coordinate system. One way to

Algorithm 1 RQS_AD: Processing Absolute Direction Query using Range Query Strategy

```
Input: rtree is the R-tree index for the dataset;
       dirRectangle is the direction region
Output: resultSet contains all object MBRs whose interiors intersect with dirRectangle

RQS_AD(R-tree *rtree, Rectangle dirRectangle) {
    currentNode = rtree;
    if overlaps(currentNode.Mbr, dirRectangle) {
        if (currentNode is a leaf node)
            Add currentNode to resultSet;
        else
            for each childNode ∈ child node of currentNode
                RQS_AD(childNode, dirRectangle);
    }
}
```

use range query strategy for object-orientation-based directional relationship is first to calculate MBR_g of the direction region, and then use this MBR as the range query constraint. The algorithm *RQS* shows the steps for object-orientation-based direction query processing. Since the MBR of the direction region is a coarse approximation of the actual direction region, the intermediate result set produced by *RQS_AD* may contain objects whose MBRs do not intersect with the interior of the direction region. A

Algorithm 2 RQS: Processing object-orientation-based direction query using range query strategy

```
Input: rtree points to the R-tree of data objects;
       referenceObject is the reference object with intrinsic orientation ;
       dirPredicate is the predicate involved in the selection condition of the query;
Output: resultSet = All object MBRs which satisfy the query

RQS(R-tree *rtree, Geometry referenceObject, Predicate dirPredicate) {
    directionRegion = computeDirRegion(referenceObject, dirPredicate);
    directionMBR = getMBR(directionRegion);
    intermediateResult = RQS_AD(rtree, directionMBR);      /* see algorithm 1*/
    resultSet = postFilter(intermediateResult);
}
```

postFilter step is needed to eliminate false hits.

Figure 5 illustrates the query processing using range query strategy(RQS). There is a reference object B with orientation as shown in figure 5. The query to be processed is *List all objects which are exactly in front of B*. The shaded region is the direction region, and the dashed rectangle is *directionMBR* which is calculated via subroutine *getMBR*. By applying algorithm *RQS_AD*, we obtain all the objects whose MBRs intersect with the interior of directionMBR. The object selected are N21, N23, N33, N42, N43, N44 as shown in Table 1. A post-filtering step is needed to exclude from intermediateResult those

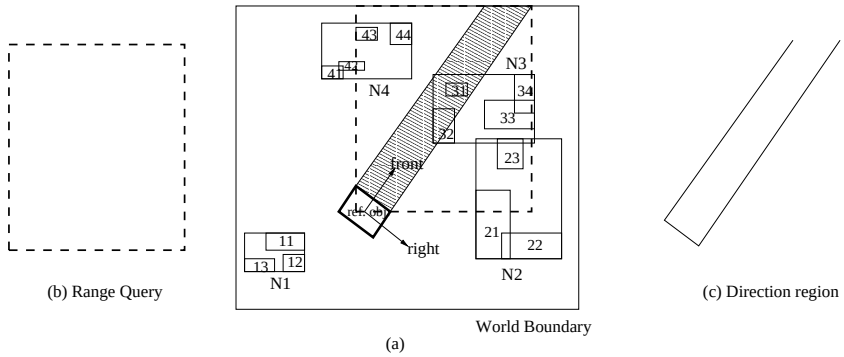


Figure 5: Processing object-orientation-based query using range query strategy

objects whose MBRs do not intersect with the interior of directionRegion. The data MBRs selected are N31 and N32. Table 1 lists the trace of the query processing. only node N1 is excluded in the first level

Steps	MBRs excluded	MBRs remain
level 1	N1	N2, N3, N4
level 2	N41, N22	N21, N23, N31, N32, N33, N42, N43, N44
postFilter	N21, N23, N33, N42, N43, N44	N31, N32

Table 1: Trace of the query processing

retrieval of R-tree, although node N2 and N4 are not satisfied with the direction predicate. There are eight object MBRs to be checked in the postFilter step and only two of them satisfy the query.

As seen from the above example, range query strategy can be used as a filter step to process object-orientation-based direction queries. However, in many instances the directionMBR is much larger than the direction region, which incurs unnecessary I/O and CPU costs. The intermediate result contains a large number of false hits. A postFilter step is required to check for intersection with the direction region, leading to additional CPU overhead.

In this paper, we propose a new strategy, namely, Open Shape based Strategy(OSS), for processing object-orientation-based direction queries. The basic idea is to use the actual direction region as the filter during traversal of spatial indices to exclude potential false hits as early as possible. This is accomplished by modeling direction regions as a new ADT, namely OpenShape, defined in terms of direction objects and operators. Open shapes refer to the geometries whose boundaries are partially defined. Open shape objects conceptually extend beyond the the boundary of embedding world. Figure 5(c) shows the open direction region. Since OSS models the direction region as an OpenShape, it does not need to know the

boundary of the embedding world, eliminating the computation related to the world boundary. OSS converts the calculation related to the direction predicates into the computation of topological operations between OpenShapes and closed geometric objects. OSS filters out irrelevant intermediate nodes during recursive search of R-tree. It eliminates the postFilter step of RQS for MBR objects. Algebraic analysis and experiment results demonstrate that the OSS outperforms RQS(range query strategy) both in terms of I/O cost and CPU cost. Relative performance of OSS in comparison to RQS is affected by several parameters including the orientation and size of the reference object, the overlap degree of the data sets, and the type of direction predicate. On average, the best relative performance improvement is achieved when angle between $front$ axis and x-axis is close to $\frac{\pi}{4} + k \times \frac{\pi}{2}$(k is an integer). Relative performance improves with the decrease in the size of the reference objects.

1.3 Scope and Outline

In this paper, we propose a new strategy to process object-orientation-based direction queries. We focus on queries that involve object-orientation-based directional relationships between simple region objects. We approximate the region objects by $MBR_g s$. A set of object-orientation-based direction predicates between region objects is defined, and our discussion about query processing is based on the predicates within this set. We assume that there is only one R-tree index with fixed orientation. The situation with multiple R-tree indices with different orientation is out of the scope of this paper.

The organization of this paper is as follows: In section 2, we define a set of direction predicates for region objects. In section 3, a new strategy OSS is proposed to process the queries involving direction predicates. A new ADT for modeling direction region is defined in section 3 as well. In section 4, an algebraic analysis for OSS is performed. The experimental evaluation is discussed in section 5. The paper ends with conclusions and recommendations for future work.

2 Object-Orientation-based Direction Predicates

In this section, we will give a brief review of the direction model and define an example set of direction predicates between region objects. More details on direction objects are available in [20].

2.1 Basic Concepts: Direction and Orientation

Direction is defined as a unit vector, i.e., a vector with magnitude equal to 1. Table 2 defines the operations on directions using vector algebra operations.

Operations	Definition
compose	$\vec{d1} + \vec{d2} = \frac{\vec{d1}+\vec{d2}}{\lvert\vec{d1}+\vec{d2}\rvert}$
deviation	$cos\theta = \vec{d_1} \odot \vec{d_2}$
reverse	$(-1) \times \vec{d1}$
isBetween[1]	$\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$ if $\exists c_1 > 0, c_2 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2}$

Table 2: Operations on Directions $\vec{d_1}, \vec{d_2}$

Operations on directions can be classified into three categories. The first category contains the operations that produce new directions: *compose* and *reverse* are operations in this category. The *compose* operation is actually achieved by *vector-addition*, and the resulting vector is scaled down by its magnitude to be a unit vector, which represents the new direction. The *reverse* operator produces the reverse direction vector. The second category of operations is to calculate the deviation between two directions. Operator *deviation* calculates the cosine of the angle between two directions, and hence gives the deviation of one direction from the other. A pair of vectors are orthogonal if their dot-product equals zero, i.e., they have 90^0 deviation. The last category of operations test the relationships among directions. The operator *isBetween* belongs to this category. In figure 6, $\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$; however, $\vec{d_1}$ is not between $\vec{d}$ and $\vec{d_2}$.

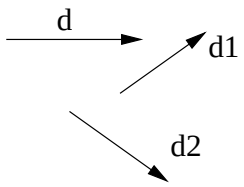


Figure 6: direction $\vec{d}$ is between $(\vec{d1}, \vec{d2})$

Orientation is modeled as a spatial object which consists of a point of origin and N pair-wise orthogonal directions, where N is the dimension of the embedding space. The origin point and the N directions form a Cartesian Coordinate System. In 2D space, the two directions may be labeled as the $\vec{right}$ and

[1] This definition works well as long as vector $\vec{d_1}$ is not parallel to vector $\vec{d_2}$. The parallel case can be handled in a user defined manner.

$\vec{front}$ directions of the orientation. Formally, we can define orientation and its operations in 2D space as follows:

Orientation is a triple O= $\langle$OP, $\vec{right}$, $\vec{front}\rangle$, where OP is a point, and $\vec{right}$, $\vec{front}$ are two orthogonal directions. It has two operations:

- $translate(O, \vec{v}) = \langle translate(OP, \vec{v}), \vec{right}, \vec{front}\rangle$;
- $rotate(O, rotationMatrix) = \langle OP, \vec{right}_n, \vec{front}_n\rangle$,
  where $(\vec{right}_n, \vec{front}_n) = (\vec{right}, \vec{front}) \odot rotationMatrix$;

An example of the rotation Matrix which rotates the orientation along the origin point for an angle θ is[17]:

$$R^{\theta}_{\vec{OP}} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Table 3 gives an illustration of these operations.

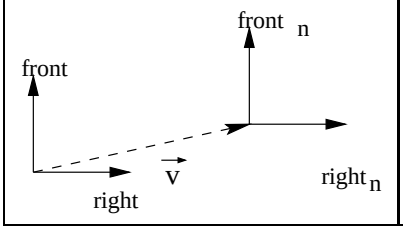
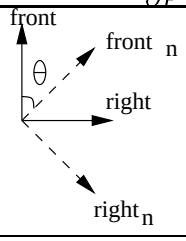
translate(O, $\vec{v}$)	rotate(O, $R^{-\theta}_{\vec{OP}}$)
	

Table 3: Operations on orientation

ADTs for direction and orientation objects are described in [20]. We will use the C++ dot notation in the rest of the paper to refer to the attributes and operations of direction and orientation objects.

2.2 Direction Predicates Between Region Objects

After reviewing direction and orientation objects and their operators, we now model object-orientation-based directions between region objects. We begin by defining an example set of direction predicates. Table 4 gives the definition of our predicates in terms of the *isBetween* operator. Left column in Table 4 lists the object-orientation-based direction predicates between regions. Right column provides a definition for the corresponding predicate using basic concepts from section 2.1. A predicate between

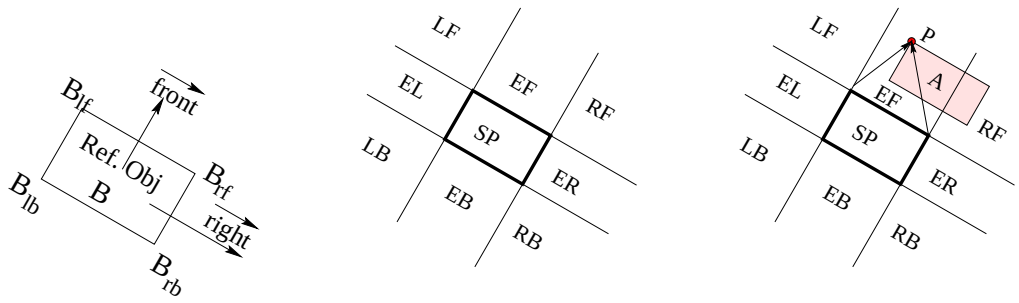
Predicates	Definitions
$SP(A,B)$	$\exists$ point P $\in$ A, s.t. P $\in$ interior(B)
$EF(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{lf}\vec{P}.isBetween(\vec{front}, \vec{right}) \wedge$ $B_{rf}\vec{P}.isBetween(\vec{front}, \vec{right}.reverse())$
$EB(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{lb}\vec{P}.isBetween(\vec{front}.reverse(), \vec{right}) \wedge$ $B_{rb}\vec{P}.isBetween(\vec{front}.reverse(), \vec{right}.reverse())$
$ER(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{rf}\vec{P}.isBetween(\vec{front}.reverse(), \vec{right}) \wedge$ $B_{rb}\vec{P}.isBetween(\vec{front}, \vec{right})$
$EL(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{lb}\vec{P}.isBetween(\vec{front}, \vec{right}.reverse()) \wedge$ $B_{lf}\vec{P}.isBetween(\vec{front}, \vec{right})$
$RF(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{rf}\vec{P}.isBetween(\vec{front}, \vec{right})$
$RB(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{rb}\vec{P}.isBetween(\vec{front}.reverse(), \vec{right})$
$LF(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{lf}\vec{P}.isBetween(\vec{front}, \vec{right}reverse())$
$LB(A,B)$	$\exists$ point P $\in$ A, s.t. $B_{lb}\vec{P}.isBetween(\vec{front}.reverse(), \vec{right}reverse())$

Table 4: Direction Predicates for MBR Objects

target region A and reference region B can be defined in terms of the directional relationship between a point inside A and the four corner points of B. It's easily explained via Figure 7. The orientation O_B of the reference object B illustrated in figure 7(a) contains two orthogonal direction objects $\vec{front}$ and $\vec{right}$ representing B's *front* and *right* directions. Figure 7(b) illustrates the corresponding direction regions for direction predicates. The directions $left$ and *behind* can be represented as $\vec{right}.reverse()$ and $\vec{front}.reverse()$ by using the *reverse* operator of the direction object. The discussions can be extended to three dimensions by adding an orthogonal axis $\vec{above}$.

The predicate set defined in Table 4 consists of nine predicates, each of which checks whether the target object A is overlapped with a specific direction region of object B with respect to B's orientation. $EF(A,B)$, $EB(A,B)$, $EL(A,B)$, and $ER(A,B)$ return TRUE if and only if object A is overlapped with the EF, EB, ER, and EL regions of B respectively as shown in figure 7. This can be formally defined using *isBetween* operator. For instance, $EF(A,B)$ is TRUE if and only if there exists a point P in A, such that, point P is within the direction region of EF. In other words, the direction $B_{lf}\vec{P}$ is between $\vec{front}$ and $\vec{right}$ and the direction $B_{rf}\vec{P}$ is between $\vec{front}$ and $\vec{right}.reverse()$ as listed in Table 4. Similarly, the predicates $RF(A,B)$, $LF(A,B)$, $RB(A,B)$, and $LB(A,B)$ will be satisfied if and only if there exists any point Q in A that Q is in the corresponding direction region. For example, $RF(A,B) = TRUE$ iff there is a point Q satisfying that $B_{rf}\vec{Q}$ is between $\vec{front}$ and $\vec{right}$.

Lemma 1 *Predicate set defined in Table 4 is complete, i.e., For any pair of simple region objects A and*



(a) Reference object

(b) Direction regions

(c) EF(A,B) iff $\exists$ a point P $\in$ A, $\vec{B_{lf}P}.isBetween(\vec{front}, \vec{right}) \wedge \vec{B_{rf}P}.isBetween(\vec{front}, \vec{right}.reverse())$

Figure 7: Direction Regions for MBR Objects, B_{lf}, B_{lb}, B_{rf} and B_{rb} represent B's left-front, left-behind, right-front and right-behind corners respectively

B with no holes, The direction of target object A with respect to the reference B can be represented in terms of a single predicate or a disjunction of several predicates in the predicate set.

Proof: As a simple convex objects without holes, the target region object A must contain at least three points that do not form a line segment. It is impossible that all the points are located on the boundary between two direction regions. In other words, there must exist at least one point located in one of the direction region. And hence, the directional relationship can be represented by the predicate that corresponding to that direction region. ■

3 Proposed Approach for Object-Orientation-based Direction Queries

We propose an open shape based strategy(OSS) using the actual direction regions modeled as open shapes. The processing of the direction predicates is then converted to the processing of the topological operations between target object and the corresponding open regions. In this section, we will first formalize the open shape based strategy and then describe new ADT of open shapes to explain the approach. This top-down description is used to simplify the presentation.

3.1 Proposed Open Shape Based Strategy(OSS)

The goal of the open shape based strategy(OSS) is to improve filtering efficiency by eliminating false hits at the earliest opportunity while recursively searching R-tree. OSS uses the actual direction region as the filter to exclude those intermediate nodes that do not satisfy the direction predicate. No refinement step is needed if objects are MBRs. Algorithm 3 shows the steps of OSS. First, OSS constructs a new

Algorithm 3 OSS: Processing object-orientation-based query using OpenShape-based Strategy

```
Input: rtree points to the R-tree of data objects;
       referenceObject is the reference object with intrinsic orientation;
       dirPredicate is the direction predicate involved in the selection condition of the query;
Output: resultSet contains all object MBRs which satisfy the query

OSS(R-tree *rtree, Geometry referenceObject, Predicate dirPredicate) {
    Object *resultSet = ∅ ;
    OpenShape dpRegion = ConstructOpenShape (referenceObject, dirPredicate);
    osQuery(rtree, dpRegion, resultSet); //find all objects interiorIntersect dpRegion
}

osQuery(R-tree *rtree, OpenShape dpRegion, Object *resultSet) {
    R-tree *currentNode = rtree;
    if dpRegion.interiorIntersects(currentNode.Mbr) {
        if (currentNode is a leaf node)
            Add currentNode to resultSet;
        else
            for each childNode ∈ child nodes of currentNode
                osQuery(childNode, dpRegion, resultSet);
    }
}
```

ADT **OpenShape** to represent the actual direction region. After obtaining *dpRegion*, OSS calls the subroutine **osQuery** with parameters *rtree* and *dpRegion* to find all objects whose interiors intersect with the interior of *dpRegion*. The subroutine **osQuery** recursively checks if the MBR of the current node intersects with *dpRegion*. The function *interiorIntersects* is designed as a member function of **OpenShape**, and returns TRUE if and only if the interior of the input object intersects with the interior of OpenShape. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, and thus eliminating the computation related to the world boundary.

Revisit the example in figure 5, given reference object B and its orientation, we want to find all objects that are EF of B. First OSS constructs the OpenShape(*dpRegion*) of the direction region represented by shadow in figure 5. After *dpRegion* is created, the second step is to check *interiorIntersects* relationships for the entries in the root of the R-tree. Only N3 remains after retrieving the first level of the R-tree,

and N31 and N32 are in resultSet. Table 5 lists the steps of query processing using OSS strategy. For MBR objects, no refinement step needs to be performed.

Steps	MBRs excluded	MBRs remain
level 1	N1, N2, N4	N3
level 2	N33, N34	N31, N32

Table 5: Trace of the query processing using OSS for the example in Figure 5

Theorem 1 *Proposed strategy OSS is complete and correct, i.e., OSS can find all objects satisfying the direction predicate and all the objects in the resultSet satisfy the given direction predicate.*

Proof: OSS first constructs *dpRegion* as an OpenShape, and then use function *dpRegion.interiorIntersects* to exclude objects whose interiors do not intersect with the interior of *dpRegion*. In other words, OSS only excludes MBRs of objects which do not intersect the interior of the corresponding direction region, namely, MBRs of the objects which do not satisfy the direction predicate. This guarantees completeness, i.e., if MBR of an object satisfies the given direction predicate, then its MBR will not be eliminated by OSS.

An object is added to the *resultSet* if and only if the MBR of the object *interiorIntersects dpRegion*. As we discussed in previous section, the interior of dpRegion represents the actual direction region satisfying the corresponding direction predicate. If the MBR of the target object A *interiorIntersects* dpRegion, MBR of A must satisfy the directional relationship. Therefore, all the MBR-objects in resultSet must satisfy the direction predicate, guaranteeing correctness. ■

We now provide the definition of new ADT **OpenShape**. OpenShape is defined as a base class hierarchy to represent open geometries whose boundaries are not closed. We focus on the data types that are needed to model open direction region. The function interfaces for **OpenShape** related to the processing of direction queries can be defined in C++ notation as follows:

```
class    OpenShape {
public:
         virtual Boolean interiorIntersects(Geometry) const;
         virtual Boolean overlaps(Geometry) const;
         virtual Boolean contains(Geometry) const;
         virtual Boolean crosses(Geometry) const;
         ⋮
};
```

In the following, we will discuss the *OpenShape* and its derived classes and use these classes to represent direction region for each direction predicate in the defined predicate set.

3.1.1 Defining OpenShape

Open shapes refer to the geometries whose boundaries are partially defined. Open shape objects have infinite interiors and extend beyond the boundary of the embedding world. Figure 8 shows some examples of open shapes that are useful for processing direction queries. Figure 8(a)-(b) are open lines with one-

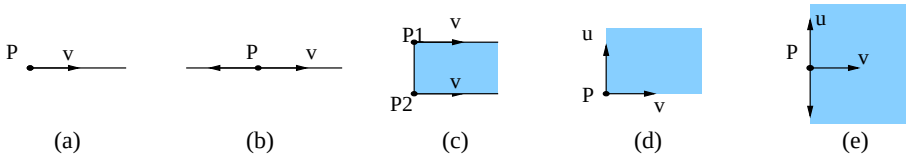


Figure 8: Examples of open shapes useful in processing direction queries

end and two-end open respectively, while (c)-(e) are open rectangles. Note that in figure 7, the direction region for each direction predicate is an open rectangle.

The *boundary* of an open shape is a set of geometries of the next lower dimension. The boundary of an one-end open line consists of the only endpoints, and the boundary of an two-end open line is empty. The boundary of an open rectangle consists of the set of sides which could be either line segments or open lines. The *interior* of an open shape consists of those points of the open shape that remains when the boundary is removed. For instance, the interior of an one-end open line is the set of points on the line except the endpoints, and the interior of a two-side open rectangle is the rectangle region which removes the boundary and indefinitely extending to the two other directions without boundary.

It would be useful if there were corresponding object classes so that users can use open shapes at abstract level. The new ADT *OpenShape* is defined as a base class of open shapes. New data types for open lines and open rectangles can be defined as derived classes of **OpenShape**, and the attributes can be described using directions and points. Table 6 defines ADT for each type of open shapes shown in figure 8. In the table, we only show the attributes and constructors of each open shape class. Examples for respective data types using parts of figure 8 are provided in the last column of the table. An OpenLine1 is defined by its endpoint and extending direction, and an OpenLine2 extending both ends and needs an intermediate point and a direction to describe it. The attributes needed to represent OpenLine1 and OpenLine2 are identical. The difference between them shows up in the definitions of various operations.

Three new data types are defined for the three subtypes of open rectangles. OpenRect1 is an one-side

ADT	attributes	constructors	examples in figure 8
OpenLine1	$startPoint : Point$; $dir : Direction$;	OpenLine1(Point, Direction);	8(a): $OpenLine1(P, \vec{v})$
OpenLine2	$interPoint : Point$; $dir : Direction$;	OpenLine2(Point, Direction);	8(b): $OpenLine2(P, \vec{v})$
OpenRect1	$vertex_1 : Point$; $vertex_2 : Point$; $dir : Direction$;	OpenRect1(Point, Point, Direction);	8(c): $OpenRect1(P_1, P_2, \vec{v})$
OpenRect2	$startPoint : Point$; $dir1 : Direction$; $dir2 : Direction$;	OpenRect2(Point, Direction, Direction);	8(d): $OpenRect2(P, \vec{u}, \vec{v})$
OpenRect3	$line : OpenLine2$; $dir : Direction$;	OpenRect3(OpenLine2,Direction);	8(e): $OpenRect3(OpenLine2(P, \vec{u}), \vec{v})$

Table 6: Abstract Data Types for Open Lines and Open Rectangles

open rectangle(e.g. figure 8c) described by two endpoints of the rectangle and one direction representing the open direction. OpenRect2 is a two-side open rectangle(e.g. figure 8d) described by an endpoint and an ordered pair of directions. OpenRect3 is a half plane or three-side open rectangle(e.g.figure 8e) and could be defined by an OpenLine2 and a direction.

3.1.2 Interpreting Direction Predicates using *OpenShape*

After defining the ADTs for open lines and open rectangles, we can construct the open direction region for each direction predicate. As illustrated in figure 7, the direction region of each object-orientation-based direction predicate is of type OpenRect1 or OpenRect2. For instance, the direction region for the EF predicate is an OpenRect1. A summary of the correspondence between direction regions and their OpenShapes is shown in table 7. Appendix **??** shows similar correspondence for point-based direction predicates.

Predicates	Type of Open Shape	Predicate true iff A $interIntersects$ the following open regions
$EF(A, B)$	OpenRect1 1-side open rectangle	$OpenRect1(B_{lf}, B_{rf}, \vec{front})$
$EB(A, B)$		$OpenRect1(B_{lb}, B_{rb}, \vec{front.reverse}())$
$ER(A, B)$		$OpenRect1(B_{rf}, B_{rb}, \vec{right})$
$EL(A, B)$		$OpenRect1(B_{rf}, B_{rb}, \vec{right.reverse}())$
$RF(A, B)$	OpenRect2 2-side open Rectangle	$OpenRect2(B_{rf}, \vec{right}, \vec{front})$
$LF(A, B)$		$OpenRect2(B_{lf}, \vec{right.reverse}(), \vec{front})$
$RB(A, B)$		$OpenRect2(B_{rb}, \vec{right}, \vec{front.reverse}())$
$LB(A, B)$		$OpenRect2(B_{lb}, \vec{right.reverse}(), \vec{front.reverse}())$

Table 7: Open shape representation for direction regions

By representing each direction region as an OpenShape, we process direction predicates using topo-

logical relationship between MBR objects and OpenShapes. This is the basic idea behind the proposed OSS strategy. When the direction region of a predicate is an open rectangle, the predicate is satisfied if and only if the target object *interiorIntersects* the corresponding open rectangle. For example, RF(A,B) return TRUE[2] if and only if A is interiorIntersected with $OpenRect2(B_{rf}, \vec{right}, \vec{front})$. We will discuss the topological operations next.

3.2 Topological Operations on Open Shapes

In this section, we will focus on defining a topological operation, namely, *interiorIntersects*, since this operation is needed to solve direction predicates as shown in table 7. We define *interiorIntersects*(open shape O, closed shape C), where O could be an OpenRect1 or an OpenRect2, and closed shape C could be a rectangle. The implementation of *interiorIntersects* uses two other topological operations of *contains* and *crosses*, which are defined in Appendix A. Other possible topological operations (e.g. *touches*, *disjoint*[12]) may be addressed in future work. In the rest of this section, O refers to an open shape, C refers to a closed rectangle. The algorithm may apply to more general cases of C such as simple convex polygon.

3.2.1 The *interiorIntersects* Operation

The *interiorIntersects* operation is defined for (OpenRect1, rectangle) and (OpenRect2, rectangle) situations. For any such pair of open shape O and closed shape C, the *interiorIntersects* relation between them is defined as:

$$O.interiorIntersects(C) = TRUE \Longleftrightarrow interior(C) \bigcap interior(O) \neq \emptyset$$

The semantics of *interiorIntersects* covers two topological operations, *contains* and *overlaps*, defined in OGIS[12] for closed geometry pairs. In other words, O.interiorIntersects(C) is true if C is contained in O, or C overlaps O. Since a closed object C cannot contain an open object O due to infinite extent, we don't have to check for C contains O. We will discuss the implementation of the OpenRect1.*interiorIntersects*(Rectangle) and OpenRect2.*interiorintersects*(Rectangle) respectively.

[2]Note that the boolean value of a topological operations between a closed shape and an OpenShape can be determined without looking at the boundary of embedding world[21].

The *interiorIntersects* Operation for OpenRect1

An OpenRect1 *interiorIntersects* a rectangle if there exists a set of points which belong to both the interior of the OpenRect1 and the interior of the rectangle. Figure 9 shows several situations that an OpenRect1 *interiorIntersects* a rectangle.

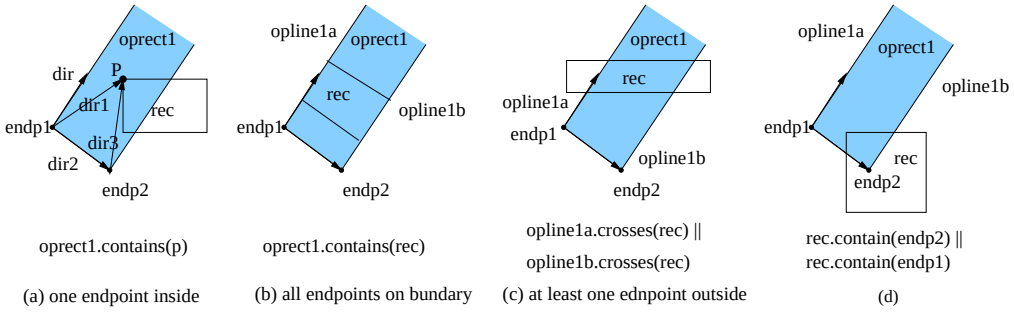


Figure 9: OpenRect1 *interiorIntersects* a rectangle

The boolean value of *interiorIntersects* operation can be determined by enumerating all possibilities of the interiorIntersecting relationship between an OpenRect1 and a rectangle. O.*interiorIntersects*(R) is TRUE if and only if at least one of the following three conditions is true:

- At least one of the endpoints of R is contained in O(e.g. figure 9(a));
- All endpoints of R are on the boundary of O(figure 9(b)).
- At least one endpoint of P is outside O(figure 9(c)) and at least one OpenLine1 of the two OpenLine1s boundary of O crosses R, i.e., OpenLine1(O.endp1, O.dir) crosses R or OpenLine1(O.endp2, O.dir) crosses R.

The first case can be tested by using *OpenRect*1.*contains*(*Point*) operation, and the second case is checked by *OpenRect*1.*contains*(*Rectangle*) operation. Both operations are described in AppendixA.2. The third case is checked by using *OpenLine*1.*crosses*(*Rectangle*) operation defined in AppendixA.1.

Efficient implementation of *interiorIntersects* operation may be other fast filter to reduce computation of above three tests. For example, if any of the endpoints of O is within R, then *interiorIntersects* operation can be determined to be true without further tests. This is illustrated in Figure 9(d). This strategy is useful for some special polygon, such as MBR, where the *MBR.contains*(*Point*) operation is cheaper. The pseudo-code for the *interiorIntersects* operation is described as Algorithm 4.

Algorithm 4 OpenRect1 *interiorIntersects* a rectangle

Input: *rec* is the rectangle that needs to be checked;
the current OpenRect1 object, which has attributes $(endp1, endp2, dir)$;
Output: $TRUE$ if interiorIntersects, $FALSE$ otherwise

```
OpenRect1::interiorIntersects(Rectangle rec) {
    /* case 1, figure 9(a) */
    for each ep ∈ endpoints of rec
        if (contains(ep))
            return TRUE;

    /* case 2, figure 9(b) */
    if (contains(rec))
        return TRUE;

   /* case 3, figure 9(c) */
    opline1a = OpenLine1(endp1, dir);
    opline1b = OpenLine1(endp2, dir);
    if ( opline1a.crosses(rec) || opline1b.crosses(rec) )
        return TRUE;

    return FALSE;
}
```

The *interiorIntersects* Operation for OpenRect2

OpenRect2.*interiorIntersects*(Rectangle) is TRUE if there exists a set of points which belong to both the interior of the OpenRect2 and the interior of the rectangle. Figure 10 shows several possible situations that an OpenRect2 *interiorIntersects* a polygon. Similar to the operation for OpenRect1, The boolean

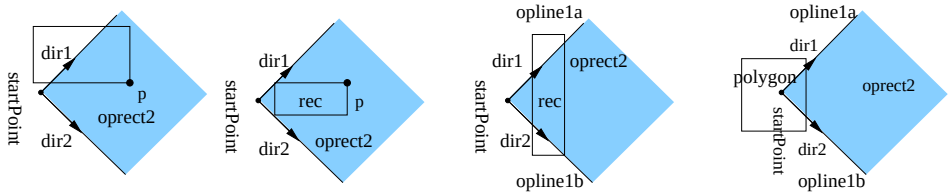


Figure 10: An OpenRect2 *interiorIntersects* a Polygon

value of $OpenRect2.interiorIntersects(Rectangle)$ operation can be determined by enumerating all possibilities of the relationship between an OpenRect2 and a rectangle. O.*interiorIntersects*(R) is TRUE if and only if at least one of the following two conditions is true:

- At least one of the endpoints of R is contained in O(e.g. figure 10(a));

- At least one endpoint of P is outside O(figure 10(b)) and at least one OpenLine1 of the two OpenLine1s boundary of O crosses R, i.e., OpenLine1(O.startPoint, O.dir1) crosses R or OpenLine1(O.startPoint, O.dir2) crosses R.

The first case can be tested by using $OpenRect2.contains(Point)$ operation described in AppendixA.2. The second case is checked by using $OpenLine1.crosses(Rectangle)$ operation defined in AppendixA.1. Note that for OpenRect2, the endpoints of any rectangle can not be all on the boundary of the OpenRect2, the above two cases are complete. Efficient implementation of $OpenRect2.interiorIntersects(rectangle)$ operation may be possible to use fast filter. figure 10(c) illustrates a cheap checking for special rectangles such as MBRs. The pseudo-code is described by algorithm 5.

Algorithm 5 OpenRect2 *interiorIntersects* Rectangle

Input: *rec* is the rectangle that needs to be checked;
the current OpenRect2 object, which has attributes $(startPoint, dir1, dir2)$;
Output: $TRUE$ if interiorIntersects, $FALSE$ otherwise

```
OpenRect2::interiorIntersects(Polygon aPolygon) {
    /* case 1, figure 10(a) */
    for each ep ∈ endpoints of aPolygon
        if (contains(ep))
            return TRUE;

    /* case 2, figure 10(b) */
    opline1a = OpenLine1(startPoint, dir1);
    opline1b = OpenLine1(startPoint, dir2);
    if ( opline1a.crosses(aPolygon) || opline1b.crosses(aPolygon) )
        return TRUE;

    return FALSE;
}
```

Lemma 2 *Algorithm* **OpenRect1::interiorIntersects** *and* **OpenRect2:: interiorIntersects** *are complete and correct.*

Proof: Figure 11 shows the flow chart of algorithm 4. The algorithm tests three cases, and it returns TRUE if and only if at least one of these three conditions is true. This guarantees the correctness of the algorithm, i.e., the algorithm returns TRUE only if the OpenRect1 *interiorIntersects* the rectangle.

Plane sweep strategy can help to prove the completeness of the algorithm. Let's consider a pair of disjoint OpenRect1 and rectangle as illustrated in figure 12(a). Move the rectangle horizontally

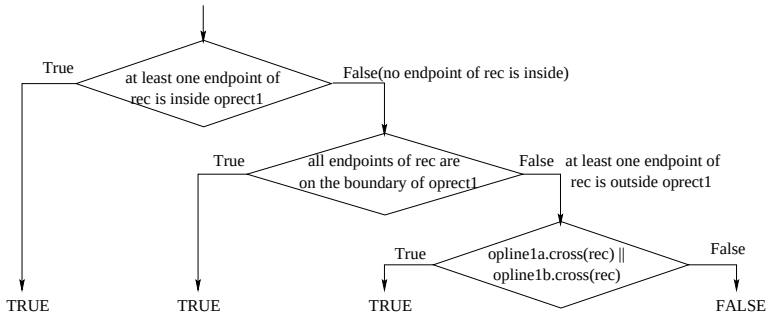


Figure 11: control flow of the algorithm

and the first case rectangle touches the OpenRect1 is illustrated in 12(b). When continuously moving the rectangle horizontally, OpenRect1 *interiorIntersects* the rectangle, and the relationship between OpenRect1 and Rectangle falls into the three cases in the flow chart. This guarantees the completeness of the algorithm, i.e., the algorithm returns FALSE only when there is no intersection between the interior of OpenRect1 and the interior of the rectangle.

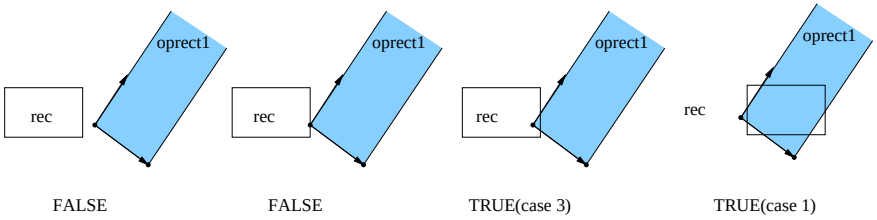


Figure 12: Possible relationships between OpenRect1 and Rectangle

The arguments of completeness and correctness for the algorithm OpenRect2::interiorIntersects are similar. ■

4 Algebraic Analysis

In this section, we perform algebraic analysis for the relative performance of the proposed open shape based strategy(OSS) vs. the classical range query strategy(RQS). We will first give a cost model for OSS and RQS, and then characterize the analysis in terms of theorems and lemmas. Since the performance is similar for the predicates whose direction regions correspond to the same OpenShape type, the analysis is performed in terms of two types of predicates, namely, the predicates whose direction regions correspond to OpenRect1 and the predicates whose direction regions correspond to OpenRect2. In the rest of the paper, we refer the first predicate type as *EF-type*, and the second as *LF-type*. Examples of EF-type

predicates include EF, EB, ER, and EL. Examples of LF-type predicates include LF,LB, RF, and RB.

I/O Cost Models

The I/O cost of a query is evaluated in terms of the number of page accesses needed in order to solve the query. RQS accesses the R-tree pages whose MBR interiorIntersects the MBR of the direction region, while OSS accesses the R-tree pages whose MBRs interiorIntersects the actual direction region. Figure

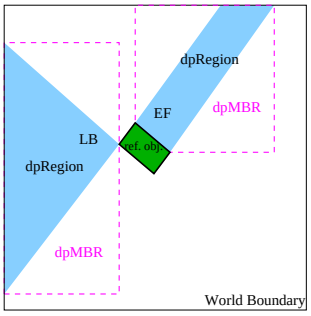


Figure 13: The ratio of RQS vs. OSS

13 illustrates the direction region and their corresponding MBRs for predicates EF and LB. The shadow regions are actual direction regions and the dashed rectangles are MBRs of the direction regions. We first make the following assumptions before we introduce the cost model:

- Data objects are uniformly distributed on embedding space.
- size of MBR objects $\ll$ size of world
- size of MBR objects $\ll$ size of direction regions

Let dpRegion(dp) represent the actual direction region for direction predicate dp, and dpMBR(dp) refer to the MBR of dpRegion(dp). Under above assumptions, the number of R-tree pages whose MBR interiorIntersects dpRegion can be approximated by the area of the dpRegion. Similarly, the number of R-tree pages whose MBR interiorIntersects dpMBR can be approximated by the area of the dpMBR. This is accurate when objects are uniformly distributed over the space and the object sizes are much smaller than the sizes of direction regions. In general these assumptions do not hold, nevertheless, this simple model is useful in understanding the relative performance of the two strategies OSS and RQS. Formally, the I/O cost is approximated as follows:

```
I/O cost(OSS, dp) ≈ area(dpRegion(dp));
I/O cost(RQS, dp) ≈ area(dpMBR(dp))=area(MBR(dpRegion(dp)));
```

The following lemmas and theories characterize the performance properties using this cost model under same assumptions.

Lemma 3 *The following statement for I/O cost is true for processing any direction predicate dp.*

$$\textit{I/O cost(OSS, dp)} \leq \textit{I/O cost(RQS, dp);}$$

Proof: By definition of MBR, for any polygon P, area(P) $\leq$ area(MBR(P)). Considering P = dpRegion(dp) for any predicate dp, we have area(dpRegion(dp)) $\leq$ area(MBR(dpRegion(dp)). ■

Lemma 4 *For any EF-type predicate dp in {EF,EB,EL,ER}, the I/O cost(OSS, dp) and I/O cost(RQS, dp) can be approximated as follows:*

$$\begin{aligned}
\texttt{I/O cost(OSS,dp)} &= \tfrac{as}{\cos\theta};\ (0^0 \leq \theta \leq 45^0) \\
&= \tfrac{as}{\sin\theta};\ (45^0 \leq \theta \leq 90^0) \\
\texttt{I/O cost(RQS,dp)} &= as\cos\theta + a^2\tan\theta;\ (0^0 \leq \theta \leq 45^0) \\
&= as\sin\theta + a^2\cot\theta;\ (45^0 \leq \theta \leq 90^0)
\end{aligned}$$

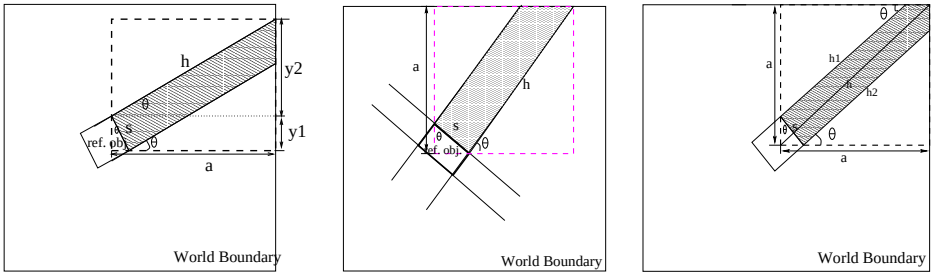
where a = the distance from the centroid of the reference object to the global boundary that intersects the dpRegion, s = size of the reference object orthogonal to the direction of dp, θ = angle between direction of dp and x-axis.

Proof: Recall the assumption that $s << a$. Assume that the reference object is at the centroid of a square-shape embedding space for simplicity. The change of orientation of the reference object leads to three different cases: *dpRegion* intersects with a boundary on X dimension(figure 14(a)); *dpRegion* intersects with a boundary on Y dimension(figure 14(b)); *dpRegion* intersects with the boundary on both dimensions(figure 14(c)). Since the reference object is located at the center of the embedding space, case 1 corresponds to the situation of $0 \leq \theta < 45^0$, case 2 corresponds to the situation of $90^0 \geq \theta > 45^0$, and in case 3, $\theta \rightarrow 45^0$. The exact boundaries of the three cases are sensitive to the ratio s/a. Since we assume $s << a$ for simplicity, case 3 can be approximated by either case 1 or case 2 for $\theta = 45^0$.

case 1(14(a)): $area(dpMBR) = a(y_1 + y_2) = a(s\cos\theta + a\tan\theta)$;
$area(dpRegion) \leq sh = \frac{as}{\cos\theta}$

case 2(14(b)): $area(dpMBR) = a(s\sin\theta + a\cot\theta) = as\sin\theta + a^2\cot\theta$;
$area(dpRegion) \leq sh = \frac{as}{\sin\theta}$

■



(a) case 1: Intersect with boundary on X dimension

(b) case 2: Intersect with boundary on Y dimension

(c) case 3: Intersect on both dimensions

Figure 14: Three cases for different orientations

Theorem 2 *For any EF-type predicate dp* $\in$*{EF, EB, EL, ER}, for a fixed size of reference object located in the center of the embedding space, the maximum I/O improvement is achieved when* $\theta = \frac{\pi}{4} + k \times \frac{\pi}{2}$*(k is an integer), while the minimum improvement is obtained when* $\theta = \frac{\pi}{2} \times k$*(k is an integer).*

Proof: According to lemma 4, the I/O cost ratio can be approximated using one of the two formulas according to the orientation of the reference object:

$$\texttt{I/O cost ratio} = \frac{I/Ocost(OSS,dp)}{I/Ocost(RQS,dp)} = \frac{a}{s}\sin\theta + \cos^2\theta; \ \ (0^0 \leq \theta \leq 45^0)$$
$$= \frac{a}{s}\cos\theta + \sin^2\theta; \ \ (45^0 \leq \theta \leq 90^0)$$

Figure 15 shows the I/O cost ratio curve for orientation degree θ in the range of $[0, 90^0]$, for some fixed values of s and a. We can notice that the maximum value of the I/O ratio is achieved when $\theta = 45^0$, and the minimum ratio is obtained when $\theta = 0^0$ and 90^0. We can also differentiate the functions for I/O

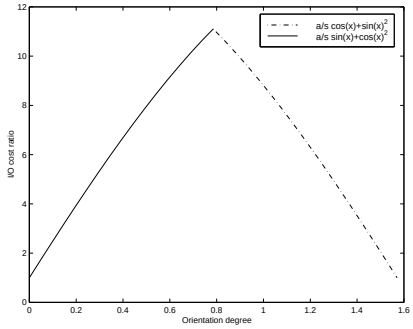


Figure 15: The I/O ratio curve for $\theta \in [0^0, 90^0]$ according to lemma 4

cost ratio to verify the maximum and minimum point.

■

Lemma 5 *For any EF-type predicate dp* $\in$*{EF, EB, EL, ER}, I/O cost ratio is affected by the size of reference objects. The smaller the object size, the larger the I/O improvement ratio.*

Proof: According to lemma 4, we have,

$$\texttt{I/O cost ratio} = \frac{I/Ocost(OSS,dp)}{I/Ocost(RQS,dp)} = \frac{a}{s}\sin\theta + \cos^2\theta;\ \ (0^0 \leq \theta \leq 45^0)$$
$$= \frac{a}{s}\cos\theta + \sin^2\theta;\ \ (45^0 \leq \theta \leq 90^0)$$

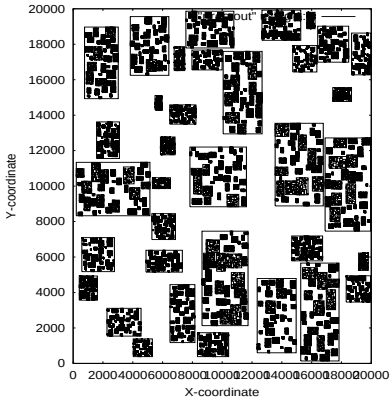
Since in the range of $0 < \theta < 90^0$, both $cos(\theta)$ and $sin(\theta)$ are greater than 0. Therefore, when parameters a and θ are fixed, the decrease of s results into the increase of I/O cost ratio.

For any predicate dp $\in$ { LF,LR,RF,RB}, the I/O cost ratio is between 1 and 2, the maximum ratio is achieved when $\theta = \frac{\pi}{4} + k \times \frac{\pi}{2}$(k is an integer). The argument for this is similar.

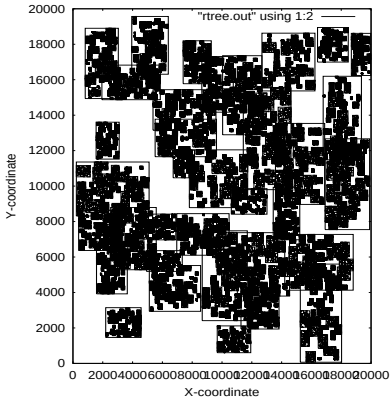
5 Experimental Evaluation

5.1 Experiment Design

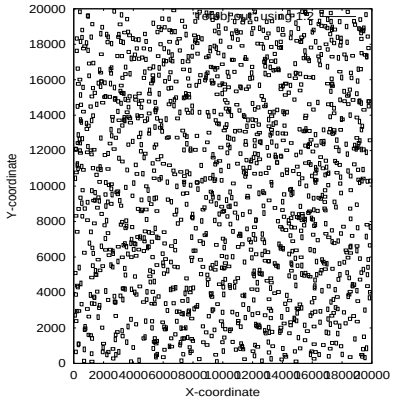
We now design an experiment to compare the performance of classical range query strategy(RQS) with the performance of open shape based query strategy(OSS) for the predicate set defined in Table 4. We use several randomly generated datasets which contain MBRs of data objects with varying sizes. The query file containing the reference object MBRs has 1500 objects. Example dataset and reference object set are shown in figure 16.



(a) Dataset R-tree without overlap among objects



(b) Dataset R-tree with overlap among objects



(c) Set of reference Object MBRs with orientation degree=90^0

Figure 16: Example of datasets

A set of queries are selected to cover different types(LF-type and EF-type) of direction predicates. LF family includes the predicates of LF, LB, RF, RB. EF family includes the predicates of EF, EB, ER, EL. On average, selectivity of EF family predicates is lower than that of LF family predicates. We consider the following queries:

- Query A_1: Given a reference object, find all objects that are EF of it.
- Query A_2: Given a reference object, find all objects that are EB of it.
- Query A_3: Given a reference object, find all objects that are ER of it.
- Query A_4: Given a reference object, find all objects that are EL of it.
- Query B_1: Given a reference object, find all objects that are LF of it.
- Query B_2: Given a reference object, find all objects that are LB of it.
- Query B_3: Given a reference object, find all objects that are RF of it.
- Query B_4: Given a reference object, find all objects that are RB of it.

The predicates in query Ai are of EF-type, and the direction predicates involved in query Bi are of LF-type. In general, query Ai has lower selectivity than query Bi.

The variable parameters for the experiment include the orientation of reference objects, overlap degree among objects in the dataset, the size of reference objects and the number of objects in the dataset. The metrics for evaluation include the number of page accesses and the number of operations required by each strategy to retrieve the query.

The orientation of a reference object is defined as the angle between $\vec{front}$ direction of the object and the $x\text{-}axis$ of the global coordinate system. When the orientation is equal to 90^0, the $\vec{front}$ direction of the reference object is parallel to the $\vec{north}$ direction in the absolute reference frame. In our experiment, orientation varies from 0^0 to 180^0.

The overlap degree(OPD) between two objects is defined as the ratio of the area of overlapping region to the area of the smaller object. The overlap degree between objects A and B is calculated as $OPD = \frac{area(A \bigcap B)}{min(area(A), area(B))}$. The overlap degree of the dataset is defined as the average overlap degree of the non-disjoint object pairs in the dataset.

The size of the reference object is measured as the relative size of the reference object to the global size. The size varies from $\frac{1}{120}$ to $\frac{1}{10}$ of the global size.

Figure 17 shows various steps of the experiment. Given the range of object size and the overlap degree, the data set is randomly generated, and the R-tree of the dataset is created based on the given branching factor. The set of reference objects is randomly generated according to its size range, which

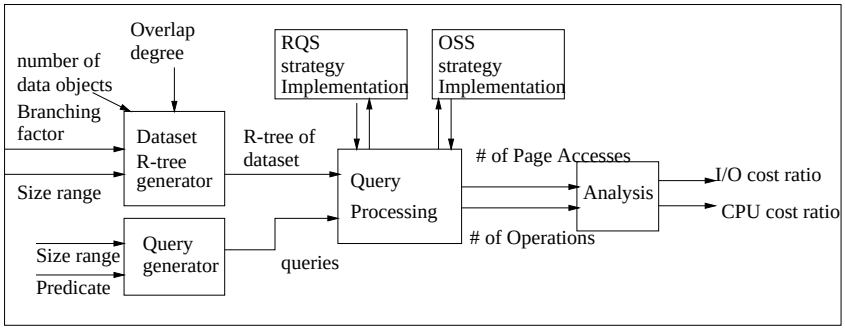


Figure 17: Experimental setup and design

can be either equal to the size range of dataset or independently given. Using R-tree and reference object set as input, "Query processing" simulates the behaviors of the RQS and OSS algorithms for different direction predicates. The total number of page accesses and the number of operations are tracked for each combination of algorithm, overlap degree, the orientation and size of reference objects and the dataset size to derive the experiment results.

5.2 Experiment Results

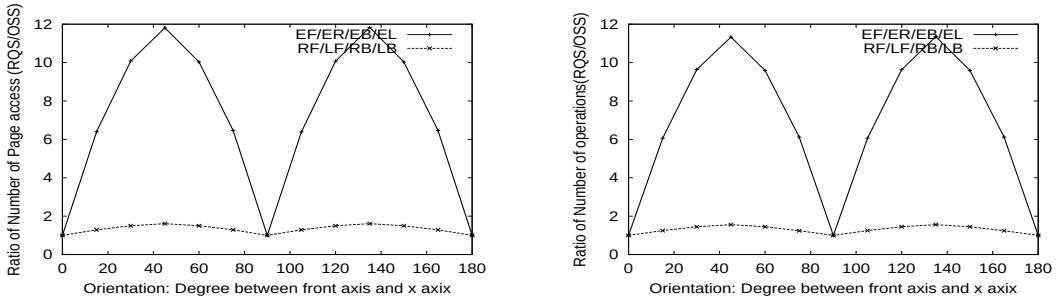
Figure 18, 19, 20 and 21 show the comparison between the classical range query strategy (RQS) and the proposed open shape based strategy(OSS) for I/O costs and CPU costs. I/O cost is measured in terms of the number of distinct pages fetched from disk. CPU cost is measured in terms of number of distinct line-line intersection operations used by an algorithm. Since the performance is similar for the predicates of same types, we only show two curves representing the relative performance of OSS and RQS for two different predicate types in each figure. The curve denoted by "EF/ER/EB/EL" illustrates the average relative performance improvement for EF-type predicates. The curve denoted by "LF/LB/RB/RB" illustrates the average relative performance improvement for LF-type predicates. For any predicate dp, the relative performance is defined as the ratio cost(RQS,dp)/cost(OSS,dp). Ratio > 1 means that OSS is cheaper than RQS.

The common observation in all figures is that OSS performs better than RQS both in terms of the number of page accesses(I/O) and the number of operations(CPU). For low selectivity predicates such

as EF-type the performance of OSS is even better compared to RQS.

5.2.1 Orientation of the reference object

Figure 18 shows the impact of the orientation of reference objects. The dataset consists of 42875 spatial objects without overlap. We use 1500 different reference objects with identical orientations, and the size of each object equals one percent of the world. The orientations of reference objects vary from 0^0 to 180^0, at an interval of 15 degrees. Recall the orientation of an object is measured by the angle between its *front* direction and the *x-axis* of the global coordinate system. Figure 18 (a) and 18(b) show the average page accesses and operations performed across 1500 reference objects respectively. OSS and



(a) Effect on Number of Page access

(b) Effect on the Number of Operations

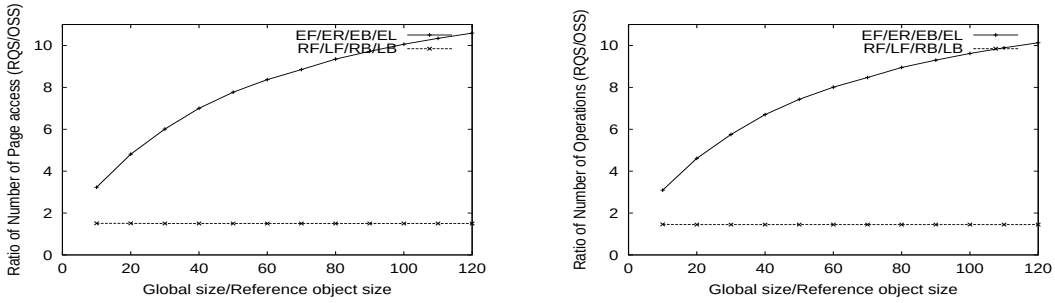
Figure 18: Effects of orientations of reference objects(orientation: $0^0, 15^0, \cdots, 180^0$, overlap degree of the data set = 0, reference object size = 1% of global size, dataset size = 42875)

RQS have the same I/O and CPU cost when the orientation is 0^0, 90^0, or 180^0 with the cost ratio equal to 1. OSS performs better than RQS for most orientations with best relative performance for OSS at 45^0 and 135^0. Relative performance advantage of OSS for predicates of EF-type varies cyclically. This result is consistent with Theorem 2 in section 4.

5.2.2 The size of reference objects

Figure 19 shows the effect of the size of the reference object on the I/O and CPU performance with fixed orientation, and fixed dataset. The dataset has 42875 objects. We use 1500 different reference objects without overlap and with fixed orientation of 30^0. The choice of orientation is arbitrary. We vary size of reference objects taking values of $\frac{1}{120}$, $\frac{1}{110}$, $\frac{1}{100}$, $\cdots$, $\frac{1}{10}$ of the total world. The *x-axis* in Figure 19 shows the ratio of $\frac{global\ size}{reference\ object\ size}$, thus the reference object size of 1% of the world corresponds to 100 in

x-axis. Note that relative gain of OSS for size = 1% is consistent with the relative gain in Figure 13 for orientation of 30^0. Obviously, OSS consistently outperforms RQS. The relative performance advantage for the EF-type predicates increases steadily with the decrease of the reference object size. This is consistent with Lemma 5.



(a) Effect on the number of page access

(b) Effect on the number of operations performed

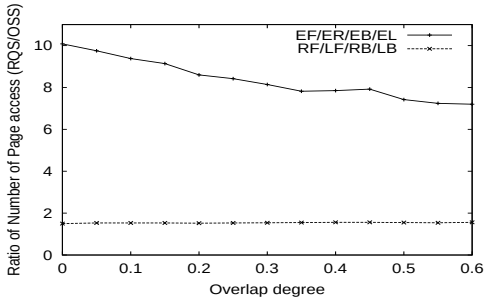
Figure 19: Effects of the size of reference objects (orientation: 30^0, overlap degree of data set =0, dataset size = 42875 objects, the size of reference object varies from $\frac{1}{120}$ to $\frac{1}{10}$ of the global size)

5.2.3 Overlap degree

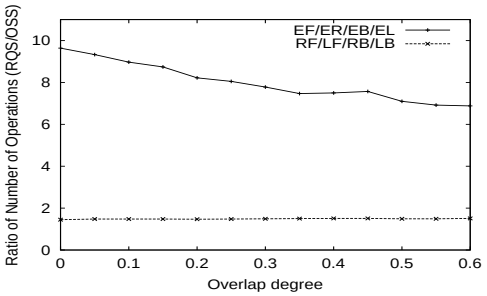
Figure 20 shows the effect of overlap degree of the dataset on the I/O and CPU performance. We used 1500 different reference object of fixed size(1%) and fixed orientation(30^0). Average overlap between objects in the data set varies from 5 percent to 60 percent. Recall that overlap degree between two objects is the ratio of common area to the smaller area of the two objects. Again, OSS consistently outperforms RQS. The relative performance advantage of OSS diminishes slightly with the increase in the overlap degree among objects. Our algebraic cost models(section 4) do not predict the effect of overlap degree. We propose to extend our model towards this purpose in future work.

5.2.4 The number of objects in dataset

Figure 21 shows the effect of the number of objects in dataset on I/O and CPU performance. We used 1500 different reference object of fixed size(1%) and fixed orientation(30^0). The objects in dataset have no overlap. The dataset size varies from 8000 to 125000. The relative performance of OSS improves with the increase in the number of objects in dataset. This result demonstrates that algorithm OSS scales better than RQS as the size of dataset increases. Our cost model does not explain why the relative



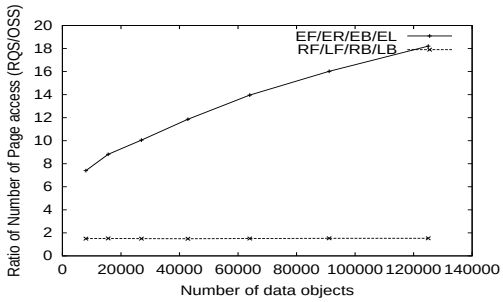
(a) Effect on Number of Page access



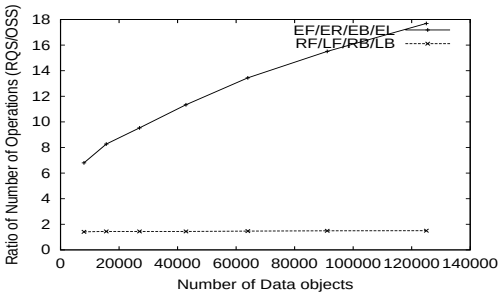
(b) Effect on Number of Operations

Figure 20: Effects of the overlap degree of the data set(Orientation = 30^0, the size of reference object = %1 of global size, dataset size = 42875, overlap degree=$0, 0.05, \cdots, 0.6$)

advantage of OSS over RQS improves with the increase in dataset size. We would like to address this issue in future work.



(a) Effect on Number of Page access



(b) Effect on Number of Operations

Figure 21: Effects of the number of objects in dataset (Orientation=30^0, Overlapping degree =0, the size of reference object = %1 global size)

6 Conclusions and Future Work

In this paper, we focus on processing queries based on object-orientation-based directional relationships. Open Shape based Strategy(OSS) is proposed for processing object-orientation-based direction queries. OSS models direction regions as *OpenShape*s, and uses actual direction region to exclude potential false hits as early as possible. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, thus eliminating the computation related to the

world boundary. The algebraic analysis and experiment results demonstrate that the OSS consistently outperforms range query strategy(RQS) in terms of both I/O and CPU cost.

In future work, we would like to consider extending OSS to other direction predicate sets[14, 6, 11]. We would also like to address the processing of queries involving viewer-orientation-based direction predicates by deriving more class and operations for ADT *OpenShape*. We would also like to investigate processing orientation-based direction queries in a mobile environment. Applying OSS to robotics and motion planning applications is another direction of future work.

Acknowledgments

This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

References

[1] D.Papadias, M. Egenhofer, and J. Sharma. Hierarchical Reasoning about Direction Relations. In *Fourth ACM Workshop on Advances in Geographic Information Systems*, pages 105–112. ACM, 1996.

[2] Yannis Theodoridis etc. Supporting Direction relations in Spatial database Systems. 30(10):1249–1257, 1996.

[3] A. Frank. Qualitative Spatial Reasoning about Cardinal Directions. In *Auto carto 10, D.Mark and D. White, eds., Baltimore, MD*, pages 148–167, 1991.

[4] C. Freksa. Using Orientation Information for Qualitative Spatial Reasoning. *Theories and Methods of Spatio-Temporal Reasoning Geographic Space*, 639:162–178, 1992.

[5] Volker Gaede and Oliver Gunther. Multidimensional Access Methods. *Computing Surveys*, 30(2):170–231, 1998.

[6] R. Goyal and M. Egenhofer. The Direction-Relation Matrix: A Representation of Direction Relations for Extended Spatial Objects . In *UCGIS Annual Assembly and Summer Retreat, Bar Harbor, ME*, 1997.

[7] G.Retz-Schmidt. Various Views on Spatial Prepositions. *AI Magazine*, 9(2):95–105, 1988.

[8] John Gurney and Elizabeth Kilpple. Composing Conceptual Structure for Spoken Natural Language in a Virtual Reality Environment . 1997.

[9] R.H. Guting. An Introduction to Spatial Database Systems. *VLDB*, 3:357–399, 1994.

[10] Antonin Guttman. R-trees: A Dynamic Index Structure For Spatial Searching. *Proceedings of ACM SIGMOD Conference*, pages 47–57, 1984.

[11] Gerd Herzog. Coping with Static and Dynamic Spatial Relations. In *Proceeding of the 5th International Workshop Time, Space and Movement*, 1995.

[12] Open GIS Consortium Inc. Opengis simple features specification for sql. http://www.opengis.org.

[13] D. Papadias and T. Sellis. Qualitative representation of Spatial Knowledge in Two-Dimensional Space. *VLDB Journal*, 3(4):479–516, 1994.

[14] D. Papadias, Y. Theodoridis, and T. Sellis. The Retrieval of Direction Relations Using R-trees. *Proceedings of the 5th Conference on Database and Expert Systems Applications(DEXA)*, 1994.

[15] Donna J. Peuquet and Zhan Ci-Xiang. An Algorithm to Determine the Directional Relationship Between Arbitrarily-shaped Polygons in the plane. *Pattern Recognition*, 20(1):65–74, 1987.

[16] P.W.Huang and Y.R. Jean. Using 2D C^+-String As Spatial Knowledge Representation For Image Database Systems. *Pattern Recognition*, 30(10):1249–1257, 1994.

[17] David F. Rogers. *Mathematical Elements for Computer Graphics.* McGraw-Hill Publishing Company, 1990.

[18] S. Shekhar, S. Ravada A.Fetterer, X.Liu, and C.T. Lu. Spatial Databases: Accomplishments and Research Needs. *IEEE Trans. Knowledge and Data Eng.*, 11(1):45–55, 1999.

[19] Shashi Shekhar, Mark Coyle, and etc. Data Models in Geographic Information Systems. *CACM*, 40(4):103–111, 1997.

[20] Shashi shekhar and Xuan Liu. Direction As a Spatial Object: A summary of Results. In *Sixth International Symposium on Advances in Geographic Information Systems*, pages 69–75. ACM, 1998.

[21] Shashi shekhar, Xuan Liu, and Sanjay Chawla. Equivalence Classes of Direction Objects and Applications. Tech. Report TR99-027, University of Minnesota, Minneapolis, MN 55455.

[22] Y. Theodoridis and D. Papadias. Range Queries Involving Spatial Realtions: A performance Analysis. *Proceedings of the 2nd Conference on Spatial Information Theory(COSIT)*, 1995.

[23] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[24] Y.I.Chang and B.Y. Yang. A Prime-Number-Based Matrix Strategy for Efficient Iconic Indexing of Symbolic Pictures. *Pattern Recognition*, 30(10):1–13, October 1997.

A Topological operations for OpenShapes

A.1 The *crosses* Operation: *crosses*(OpenLine O, Closed shape C)

The *crosses* operation applies for object pair between an open line O and a closed shape C, where O could be OpenLine1 or OpenLine2 and C could be points, line segments and polygons. For any such pair of O and C, the *crosses* relation between them is defined as:

$O.crosses(C) = TRUE \Longleftrightarrow O.\vec{dir} \odot Direction(C - O.startPoint) = 1$; (if C is a point)

$\Longleftrightarrow \exists$ a point P s.t. $P \in interior(O) \cap interior(C)$;(if C is not a point)

Figure 22 shows some examples of the *crosses* operations. We approximate polygonal closed shape by rectangle for simplicity, however, the algorithm will work for convex polygon as well. Figure 22(a) shows an OpenLine1 O crosses a point C, figure 22(b) shows O crosses a line segement C, and Figure 22(c) consists of the five possible cases that an OpenLine1 crosses a polygon C. To implement the *crosses* operation, we use vector dot-product and the *isBetween*

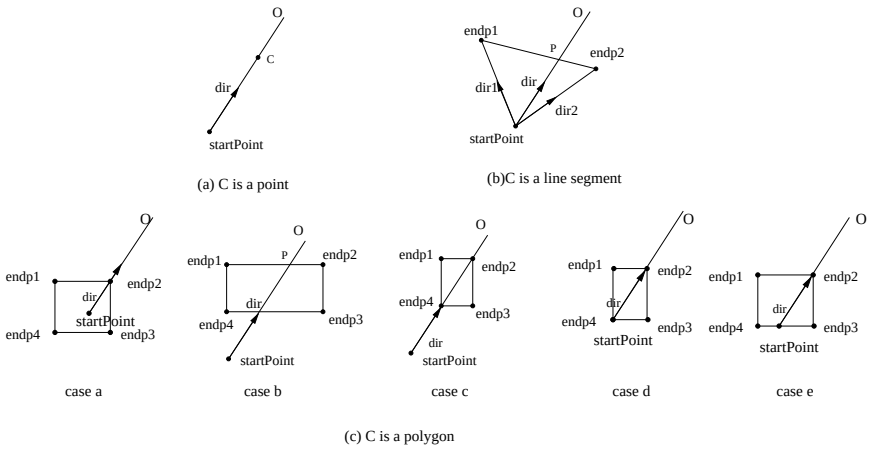


Figure 22: OpenLines crosses Point/Line/Rectangle

operation of direction objects. For the case of OpenLine1 crosses a given point, such as in figure 22(a), the OpenLine1 O *crosses* the given point C if and only if the vector-product between $O.\vec{dir}$ and the direction $Direction(C - O.startPoint)$ equals 1. For the case of OpenLine1(O) *crosses* a line(C) as in figure 22(b), $O.\vec{dir}$ should be between the two directions constructed by the two endpoints and the O.startPoint, i.e. $\vec{dir}1$ and $\vec{dir}2$ in the figure. The *crosses* relation between an OpenLine1 and a polygon is more complicated for there are several situations possible as shown in figure 22(c). We solve them by enumerating all possible situations that O.crosses(C) will be TRUE. Algorithm 6 describes pseudocode of OpenLine1::crosses. The topological operations *contains* and *touches* for closed geometry pairs defined by OGIS [12] are used in the algorithm for checking if O.startPoint is in the interior or on the boundary of the polygon.

Lemma 6 *Algorithm* **OpenLine1::crosses** *is complete and correct.*

Algorithm 6 OpenLine1 Crosses Points/Lines/Polygons

Input: closeShape is the topologically closed object that needs to be checked;
the current OpenLine1 object, which has attributes $(dir, startPoint)$;
Output: $TRUE$ if crossed, $FALSE$ otherwise

```
OpenLine1::crosses(Point closeShape) {
    dir1 = Direction(closeShape − startPoint);
    if (dir.deviate(dir1))
        return TRUE;
    return FALSE;
};
OpenLine1::crosses(Line closeShape) {
    dir1 = Direction(closeShape.endp1 − startPoint);
    dir2 = Direction(closeShape.endp2 − startPoint);
    if (dir.isBetween(dir1, dir2))
        return TRUE;
    return FALSE;
};
OpenLine1::crosses(Polygon closeShape){
    for each side ∈ sides of closeShape
        if crosses(side)
            return TRUE;
    /* The following deal with the cases O does not cross any side of the polygon */
    if (closeShape.contains(startPoint))        /* figure 22(c) case a: startPoint in the polygon */
        return TRUE;
    if (closeShape.touches(startPoint)) {
        if startPoint ∈ endpoints of closeShape {        /* figure 22(c) case d */
            for each ep ∈ endpoints of closeShape
                if crosses(ep) && !adjacent(startPoint, ep)
                    return TRUE;
        } else {        /* figure 22(c) case e: startPoint is on one side of the polygon */
            for each ep ∈ endpoints of closeShape
                if crosses(ep) && ep ∉endpoints of the side that crosses startPoint
                    return TRUE; }
        return FALSE;
    }
    /* figure 22(c) case b and c : startPoint is outside of the polygon */
    for each ep ∈ endpoints of closeShape {
        if crosses(ep) {
            for each ep2 ∈ endpoints of closeShape -{ep}
                if (!adjacent(ep,ep2) && crosses(ep2))
                    return TRUE;     /*22(c) case d: O crosses non-adjacent endpoints.*/
        break; }
    return FALSE;
};
```

Proof: The completeness and correctness of the algorithm OpenLine1::crosses for Point and Line are proved by definition. In the following, we will demonstrate that OpenLine1::crosses(Polygon) is complete and correct.

- Completeness:
  For an OpenLine1(O) and a Polygon(C), there are two possibilities:
  - O crosses at least one of the sides of the polygon(figure 22(c) case b) $\Longrightarrow$ O.crosses(C) is TRUE;
  - O doesn't cross any side of C. The algorithm solves this possibility by considering the location of the startPoint.
    1. O.startPoint is within the polygon(figure 22(c) case a) $\Longrightarrow$ O.crosses(C) is TRUE ;
    2. O.startPoint is outside C, O.crosses(C) iff O crosses two non-adjacent endpoints of C(figure 22(c) case c) ;
    3. O.startPoint is on the boundary of C:
       Case c: O.startPoint is an endpoint of C, and O crosses an endpoint *ep* which is not adjacent to O.startPoint $\Longrightarrow$ O.crosses(C) is TRUE
       case d: O.startPoint falls on side *s* of C, O.crosses(C) is TRUE iff O crosses endpoint *ep* of C and $ep \notin$ endpoints of *s*.

  For any other situation that is not described above, OpenLine1 does not cross the polygon. Since the algorithm *crosses* checked all the above situations, the algorithm will find all the *crosses* relation between an OpenLine1 and a polygon. Therefore, the algorithm is complete.
- Correctness:
  The algorithm returns TRUE if and only if one of the situation is satisfied, in which case, the OpenLine1 crosses the polygon. The algorithm only returns TRUE when the operation *crosses* hold, therefore, the algorithm is correct. ■

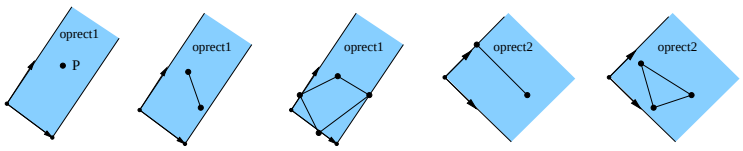
The *crosses* operation between an *OpenLine*2(two-end openline) opline2 and a closed shape can be implemented by converting the opline2 to two *OpeLine*1s: opline1a and opline1b. The *startPoint* of each *OpenLine*1 is the *interPoint* of opline2, and the *dir* of each *OpenLine*1 is *opline*2.*dir* and *opline*2.*dir.reverse*() respectively. The opline2 crosses the closed shape if and only if either opline1a or opline1b crosses the object.

A.2 The *contains* Operation for Open Rectangles

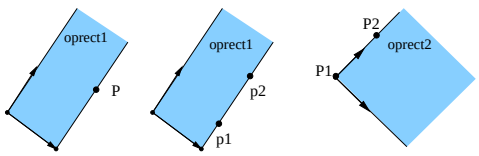
For any open rectangle O and closed shape C, $O.contains(C)$ returns TRUE if and only if C is wholly contained within O. O could be an OpenRect1 or an OpenRect2, and C could be a point, a line segment, or a polygon. The formal definition for O.contains(C) is :

$$O.contains(C) = TRUE \Longleftrightarrow (O \cap C = C) \text{ and } (interior(O) \cap interior(C) \neq \emptyset)$$

Figure 23 shows some examples of the *contains* operations. Figure 23(a) are exmples that satisfy *contains* relationship,



(a) Examples that satisfy O.contains(C)



(a) Examples that does not satisfy O.contains(C)

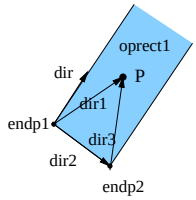
Figure 23: Examples of contains relationship

where figure 23(b) are examples that C is on the boundary of O, but not contained in O.

Since we are only interested in the operations that are used for directional queries, we restrict our discussion in the cases where O is an OpenRect1 or OpenRect2, and C is a point or a polygon.

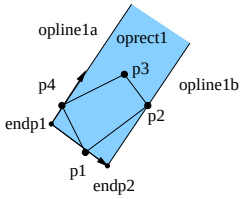
The *contains* Operation for OpenRect1

OpenRect1 is an one-side open rectangle. Figure 24 shows the examples that an OpenRect1 contains a point/polygon. The implementation of the *contains* operation can be accomplished by using *isBetween* operator and the *crosses* operation.



dir1.isBetween(dir,dir2) &&
dir3.isBetween(dir, dir2.reverse())

(a) C is a point



any endpoint is either on the boundary or contained in the oprect1

(b) C is a polygon

Figure 24: OpenRect1 contains point/line/polygon

In order to check if a point is contained in OpenRect1(figure 24a), two direction objects are constructed by substracting the point C from each endpoints of the OpenRect1, and $isBetween$ operations is applied.

For any convex polygon C, if there does not exist any endpoint that is outside the OpenRect1, i,e., every endpoint of C is contained in the OpecRect1 or located on the boundary of the OpenRect1, then C is contained in the OpenRect1. Algorithm 7 describes the operation in pseudocode.

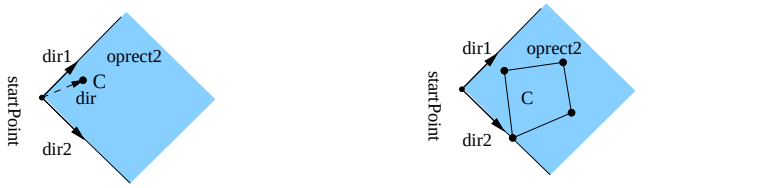
Algorithm 7 OpenRect1 contains Point/Polygon

Input: closeShape is the topologically closed object that needs to be checked;
the current OpenRect1 object, which has attributes $(endp1, endp2, dir)$;
Output: $TRUE$ if contain, $FALSE$ otherwise

```
OpenRect1::contains(Point closeShape) {
    dir1 = Direction(closeShape − endp1);
    dir2 = Direction(endp2 − endp1);
    if (dir.isBetween(dir1, dir2)) {
        dir3 = Direction(closeShape − endp2);
        if dir3.isBetween(dir,dir2.reverse())
            return TRUE;
    }
    return FALSE;
};
OpenRect1::contains(Polygon closeShape){
    opline1a = OpenLine1(endp1, dir);
    opline1b = OpenLine1(endp2, dir);
    aLine = Line(endp1, endp2);
    for ep ∈ endpoints of closeShape
        if !(endp1 == ep || endp2 == ep || contains(ep) ||
                opline1a.crosses(ep) || opline1b.crosses(ep) || aLine.crosses(ep))
            return FALSE;
    return TRUE;
};
```

The *contains* Operation for OpenRect2

OpenRect2 is a two-side open rectangle, which is described by one endpoint and two directions. Figure 25 shows several situations that an OpenRect2 contains a Point or a Polygon. Similar to the case of OpenRect1, the $isBetween$ operation of the direction objects is used to implement the *contains* operation between an OpenRect2 and a point. A convex polygon is contained in the OpenRect2 if every endpoints of the polygon is either contained in the OpenRect2 or on the boundary. The pseudocode is described in algorithm 8.



(a) C is a point

(b) C is a polygon

Figure 25: OpenRect2 contains point/line/polygon

Algorithm 8 OpenRect2 contains Point/Line/Polygon

Input: closeShape is a topologically closed object;
the current OpenRect2 object, which has attributes $(startPoint, dir1, dir2)$;
Output: $TRUE$ if contain, $FALSE$ otherwise

```
OpenRect2::contains(Point closeShape) {
    dir = Direction(closeShape − startPoint);
    if (dir.isBetween(dir1, dir2))
        return TRUE;
    return FALSE;
};

OpenRect2::contains(Polygon closeShape){
    opline1a = OpenLine1(startPoint, dir1);
    opline1b = OpenLine1(startPoint, dir2);
    for ep ∈ endpoints of closeShape
        if !(ep == startPoint || contains(ep) || opline1a.crosses(ep) || opline1b.crosses(ep))
            return FALSE;
    return TRUE;
};
```

Completeness and Correctness of the algorithms

Lemma 7 *Algorithm* **OpenRect1::contains** *and* **OpenRect2::contains** *are complete and correct.*

Proof:

- Completeness:
  For object pair of OpenRect1(O) and Point(C), there is only one case that O.contains(C) is true, that is C is in the interior of O. The completeness of the algorithm is clear by definition. For the case of checking OpenRect1.contains(Polygon), the algorithm 7 excludes only the cases that there exist some endpoints of the polygon outside the OpenRect1, when the polygon is absolutely not wholly contained in OpenRect1. Therefore, the algorithm will find all cases when the relationship *contains* is satisfied. The clgorithm find all possible contain relations, so is complete.
- Correctness:
  The algorithm OpenRect1.contains(Point) returns TRUE if and only if the point is contained in the interior of the OpenRect1. For the object pair of an OpenRect1 and a convex polygon, the algorithm returns TRUE if and only if every endpoint of the polygon is either contained or on the boundary of the OpenRect1, in which case, there does not exist any point in the interior of the polygon which is not contained in the OpenRect2. therefore, the algorithm will return TRUE only when polygon is wholly contained in OpenRect1.

The argument of the completeness and correctness for algorithm 8 is similar. ■